Toward Specifying and Validating Cross-Domain Policies*

Michael Hicks
Department of Computer Science and UMIACS
University of Maryland, College Park
mwh@cs.umd.edu

Nikhil Swamy
Department of Computer Science
University of Maryland, College Park
nswamy@cs.umd.edu

Simon Tsang
Telcordia
stsang@research.telcordia.com

April 17, 2007

Abstract

Formal security policies are extremely useful for two related reasons. First, they allow a policy to be considered in isolation, separate from programs under the purview of the policy and separate from the implementation of the policy's enforcement. Second, policies can be checked for compliance against higher-level security goals by using automated analyses. By contrast, ad hoc enforcement mechanisms (for which no separate policies are specified) enjoy neither benefit, and non-formal policies enjoy the first but not the second.

We would like to understand how best to define (and enforce) multi-level security policies when information must be shared across domains that have varying levels of trust (the so-called "cross domain" problem). Because we wish to show such policies meet higher-level security goals with high assurance, we are interested in specifying cross domain policies formally, and then reasoning about them using automated tools. In this report, we briefly survey work that presents formal security policies with cross-domain concerns, in particular with respect to the problem of *downgrading*. We also describe correctness properties for such policies, all based on *noninterference*. Finally, we briefly discuss recently-developed tools for analyzing formal security policies; though no existing tools focus on the analysis of downgrading-oriented policies, existing research points the way to providing such support.

1 Introduction

Cross-domain security is concerned with the transmission of information between different administrative domains, which may have varying levels of mutual trust. As examples, cross domain flows occur when a U.S. Army commander shares information with trusted coalition partners (e.g. UK, Canada), less trusted coalition partners (e.g. forces from non-NATO countries), and even non-Army, U.S. Government entities (e.g. FBI, IRS).

It is usually reasonable to characterize a cross-domain information flow as one which reaches a less trusted party with respect to the multi-level security (MLS) policy of the originator. Such a flow should

*University of Maryland Dept. of Computer Science Technical Report # CS-TR-4870 and UMIACS Technical Report # UMIACS-TR-2007-23. Prepared through collaborative participation in the Communications and Networks Consortium sponsored by the U. S. Army Research Laboratory under the Collaborative Technology Alliance Program, Cooperative Agreement DAAD19-01-2-0011. The U. S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

only be allowed if the information is *downgraded* to remove any sensitive content [SS05]. For example, a U.S. commander might wish to share information with a non-U.S. coalition commander about the location of an enemy camp, but only after removing from the document the source of the information. This removal constitutes a downgrading of the information.

Downgrading, and its importance, has long been recognized, but it is a process that largely happens *outside* of a formal security policy analysis. That is, MLS policies indicate which flows are allowed, and any flows not allowed by the policy can be allowed by overriding the policy, usually after a manual audit. The disadvantage of this is that such "out-of-band" actions preclude assessment by analyzing the policy. As an example, in security-oriented programming languages (SPLs) [SM03] like Jif [MNZZ01], downgrading is enabled through *selective declassification* in which the programmer annotates some code to indicate that the policy that would otherwise be enforced can be overridden in this case (so-called *robust declassification* [ZM01, MSZ04] is similarly tied to the code). This confounds reasoning about the effects of policy in isolation; rather both the policy *and the programs using the policy* must be considered.

This report consists of two parts. The first part reviews prior work that has sought to specify a notion of downgrading as part of a formal multi-level security (MLS) policy. In particular, we consider Chong and Myers' *declassification policies* [CM04]; the *trusted declassifiers* [HKMH06] of Hicks et al., and Li and Zdancewic's *downgrading policies* [LZ05a, LZ05b]. This report also considers the security goals (or "correctness conditions") appropriate for downgrading-aware policies. Since *noninterference* [GM82, GM84] is a standard goal for MLS policies, the above authors have defined properties that relax noninterference in some way to accommodate downgrading.

The second part considers prior work in the automated analysis of formal security policies. While we are not aware of prior work on the analysis of downgrading-aware policies, prior approaches would be a starting point for further research.

2 Security Policies and Goals for Downgrading

This section summarizes prior research in specifying security policies that accommodate downgrading, the core element of cross-domain policies.

2.1 Declassification Policies

Chong and Myers [CM04] have defined declassification policies p as security labels having the form $l \longrightarrow^{c} p$. This label consists of an initial security label l that can be declassified to policy p when condition c is true. Thus the labeling for a piece of data indicates how its policy might change in the future, and under what conditions. Such a policy could be used in several ways. For example, the label $S \longrightarrow^{curryear \geq 2010} UC$ could express that a document currently labeled as *secret* (S) should be relabeled to be *unclassified* (UC) after the year 2010. Another example might be to label a document s with $S \longrightarrow^{\texttt{isAnonymized}(s)} UC$, which states that s is labeled *secret* until some function `isAnonymized` indicates it is safe to treat as *unclassified*.

Chong and Myers define a simple programming language that makes use of declassification policies in conjunction with an expression form for selective declassification: `if declassify(`e`) then` s_1 `else` s_2, where e is the data to be declassified. The type checker ensures that the declassification will only succeed (and thus execute s_1) if e has policy $l \longrightarrow^{c} p$ and condition c is satisfied at the point of the declassification; otherwise the declassification fails and s_2 is executed. In essence, this adds a level of checking to selective declassification, preventing the declassification from taking place if conditions are not yet right.

For this language Chong and Myers prove a property called *noninterference until* $c_1, ...c_k$. When applied to some program variable x, this property states that a type correct program is non-interfering (i.e., it adheres to a standard MLS policy) on the variable x at some label l until the program has performed k declassifications due to the conditions $c_1, ..., c_k$. In essence, this states that the program will not release information until conditions set forth in the policy are met.

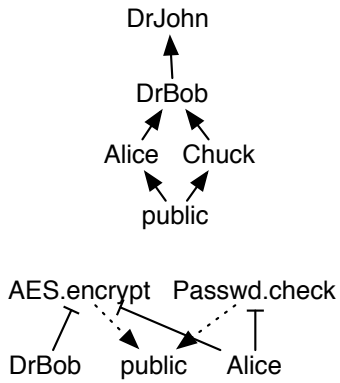

Figure 1: A security policy, consisting of a security lattice (top) and trusted declassification specifications (bottom). Combining the two induces a graph of legal flows between principals.

2.2 Trusted Declassifiers

The problem with selective declassification, even when combined with declassification policies, is that a security analyst must look at the code of an application to understand the impact of declassification operations on the overall policy goals. The insight behind trusted declassifiers [HKMH06] is that declassification operations can be mentioned in a policy totally separate from programs that use them if (1) declassification operations have a *name*, and (2) if the programmer has defined abstract conditions under which the named declassification operations can be used. This information is sufficient to perform policy analysis and to permit proof that the program adheres to the policy via type checking or static analysis (as with most SPLs).

Hicks et al. [HKMH06] show how to do this by defining declassification operations as methods implemented in an object-oriented programming language. In addition to the MLS security lattice,[1] the policy indicates which declassification methods are trusted by which principals. This is illustrated in Fig. 1. The top of the figure shows the *principal hierarchy* ordered according to the arrows—principals higher in the lattice can read documents visible to principals lower in the lattice. Thus `DrJohn` is permitted to read the same documents as all other principals, while `public` has the least privilege. The bottom of the figure indicates the declassifiers trusted by certain principals, and to what security level those declassifiers can downgrade their input data. For example, `DrBob` trusts the `AES.encrypt` method to downgrade his data to be visible by `public`, and likewise for Alice, who also trusts `Password.check` to downgrade her data to public.

The new policy illustrates flows allowed by the system due to the "normal" policy (lattice) and due to downgrading. For example, we can see that `DrBob`'s data could be viewed by `public`, but only after it has been encrypted by `AES.encrypt`. In a system with selective declassification, this flow would only be visible by inspecting program code along with the traditional policy shown at the top of Figure 1.

Moreover, the SPL can check that the only downgrading that is allowed is according to these declassifiers, and thus prove that the policy is properly enforced. Hicks et al. extend the Jif [MNZZ01] programming language with support for trusted declassifiers. They prove that a simpler, formal language outfitted with trusted declassifiers, called *FJifP*, enjoys a property called *noninterference modulo trusted declassifiers*. In the degenerate case in which there are no trusted declassifier uses in the program, the property is equivalent to noninterference. Otherwise the property makes clear that any information leakage must be due to the

[1] A security lattice consists of a set of principals ordered according to their privilege. For example, the standard military lattice is $TS > S > C > UC$ where TS stands for *top secret*, S stands for *secret*, C stands for *confidential*, and UC stands for *unclassified*. The lattice states, when viewed from the point of view of confidentiality policies, that principals at level TS have greater privilege (can read more documents than) principals at level S, and so on for the other levels.

trusted declassifiers allowed by the policy.

2.3 Downgrading policies

Li and Zdancewic [LZ05a] propose *downgrading policies* which, similar to Chong and Myers' declassification policies, are expressed as labels on data. These labels indicate *how* information about the data can be leaked. In particular, a label consists of a function whose input parameter represents secret data, and whose result can be considered public. In effect, the label of some data indicates precisely by what computation it may be leaked, and the enforcement mechanism can ensure that any program actions on data will not leak information in violation of the policy of its label.

As an example, the notation $\lambda x.x$ describes a function with a single argument x that immediately returns that argument—the so-called "identity function."[2] This function can be treated as the label `public`, since as "input" it takes a secret value x, and then returns that value, making it publicly-visible. At the opposite extreme would be a label consisting of the set of constant functions $\lambda x.c$ (where c is some constant integer); this label indicates that no matter what secret argument is passed in, the same constant value comes out. One can think of this as like a password checking algorithm that always says "no!" The middle ground would be a label such as $\lambda x.(x = p)$ where p is a (secret) password and $=$ represents the equality operation (not assignment). This says that information about x can be made public by comparing it to some password p; in other words, a public user will learn, each time the password checker is invoked, whether a value x is or is not p. Since this leaks a small amount of information about p, it expresses a kind of downgrading.

Li and Zdancewic prove that their system enjoys a property they call *relaxed noninterference*, which essentially states that the only information leaked is that according to the policy [SM04]. They have also adapted their approach to integrity policies [LZ05b].

2.4 Discussion

Downgrading policies bear some resemblance to both declassification policies and trusted declassifiers. Like the first, they are expressed as labels on data; the difference is that Chong and Myers' condition c is fleshed out as a downgrading function that transforms the data, rather than just permitting it to be released. This is similar to a trusted declassifier, but is more concrete; a trusted declassifier is identified by its name and some arbitrary code (its implementation). On the other hand, trusted declassifiers are in some ways more general because their inputs and outputs can be arbitrary levels in the security lattice, whereas for downgrading policies the input is always secret and the output is always public (i.e., the system assumes a two-point lattice with `secret` > `public`). Both downgrading and declassification policies suffer from the fact that flows are indicated by security labels, which are part of programs, as opposed to a policy specified independent of the data in any given program.

The noninterference-based security properties satisfied by programs using the proposed policies of the three systems are similar. They all state that as long as downgrading is not used then noninterference holds. Relaxed noninterference states that the information leakage exactly matches the downgrading policy, which in turn indicates exactly how information is downgraded. Trusted declassifiers are treated more abstractly, but information loss can also be attributed only to them. Chong and Myers' "noninterference until $c_1, ..., c_k$" indicates under what circumstances a declassification will occur (for a particular variable).

Swamy et al. define an approach for proving the proper enforcement of MLS policies that are allowed to change over time [SHTZ06]. The security property proven for this approach is similar to the "noninterference until $c_1, ..., c_k$" property of downgrading policies. In particular, it shows that (1) the execution is non-interfering until the policy is updated; (2) the act of updating the policy itself does not leak information; and (3) after the policy has been updated all subsequent flows are consistent with the new policy.

One unsatisfying aspect of the noninterference-based security properties supported by these policies is that they are not *quantitative*. In particular, they indicate that a particular variable could leak information

[2]The formal notation $\lambda x.e$ defines a function with formal parameter x and body e. When the function is called, e will execute with x replaced by the actual argument. The value that results from this execution is returned to the caller.

under certain conditions, but they do not indicate how often this variable is leaked, i.e., the quantitative rate of declassification. Understanding how to reason about quantitative information flow is in its early stages (e.g., see recent work by Malacaria [Mal07]).

3 Formal policy analysis

Being able to prove that programs meet higher level security goals, like noninterference, is extremely useful. However, these goals depend on the security policy itself being correctly specified. For example, an analyst may wish to understand whether a policy mistakenly allows a flow from principal A to principal B toward satisfying a higher-level security goal.

To perform such an analysis, we must understand the meaning of a policy, which essentially derives from how it is to be enforced. Broadly speaking, overall security enforcement uses three elements:

1. A definition of roles, which group together principals with similar privileges. Role definitions might be expressed in a declarative policy language like RT (which has a notion of administrative model, which is useful) or RBAC, or encoded in a traditional security lattice like the ones mentioned above (e.g., using a principal to designate a role).

2. A mapping of roles to objects and the operations on objects; this is called a "labeling" in the preceding discussion. So far we have been discussing operations on objects that relate to information flow: reads and writes. We might like to reason about operations more generally to support other security goals.

3. Enforcement of the above two elements in an actual program. For example, we want to be sure that when the program is acting on behalf of principal P, it can only perform operation M on object O if the object's label allows M for roles R for which P is a member. In the preceding discussion, the cross-domain policies we considered were defined for security-oriented programming languages (SPLs), and the enforcement is performed by the programs themselves, ensured by the type checker. In other words, if the program in question type checks, then we are sure that it properly enforces its policy.

In short, the first two elements define the policy, and the last element ensures proper enforcement of the policy. What we would like, given this, is to be able to reason about the effects the policy (i.e., that the policy itself does not permit actions or flows that we don't expect) and ensure the enforcement of the policy in the program leads to a higher level security goal (information flow, access control, separation of duty, etc.). In the remainder of this section we focus on the former; we will consider the latter in ongoing work (using SPLs).

Past work takes two basic approaches to analyzing policies: (1) encode policies and their queries as logic programs; (2) encode policies as models and implement queries using model checking. In these settings, queries typically consist of, broadly speaking: (1) *Verification*: does the policy prevent (allow) access by some subject to a particular object? (2) *Change analysis*: if the policy is changed according to some specification, are certain invariants about the policy (in terms of verification queries) maintained?

3.1 Encoding Policies as Logic Programs

A typical approach toward reasoning about policies is to formalize the policy as a logic program and then to formulate queries using the logic programming language. For example, the semantics of the RT_0 role-based specification language [LMW02] is given using the relational language Datalog, a cousin of Prolog. Each RT_0 rule can be encoded as a Datalog predicate. Using such an encoding, users can write small Datalog policies to query, for example, whether a principal is a member of a particular role. Li et al. [LMW05] have also looked at implementing queries that characterize potential changes to a policy. For example, one might ask whether a particular principal will still be considered a member of a role if a particular rule is removed. Other elements of the RT family of languages are specified in terms of Datalog augmented with constraints [LM03], due to their increased expressiveness. In these languages the change analysis problem becomes undecidable [LMW05].

There are many other examples in this basic style. Hicks et al. [HRC+07] define a Prolog logic programming-based semantics for SELinux MLS policies, and use this semantics to prove overall properties about the policy, e.g., that the polices enforce the Bell-LaPadula [BL75] $*$-property. Sarna-Starosta and Stoller [SSS04] use logic programming specify the type enforcement portion of SELinux policy, and are able to perform information flow queries (e.g., show that a standard user's read access to the raw disk type `fixed-disk-device-t` is only permitted via the file system administrator type `fsadm_t`), integrity queries (e.g., the dual to the read query above), separation of duty queries (e.g., to prove that system daemons allowed to execute programs are not allowed to write or create those programs), and completeness queries (e.g., to find all permissions neither explicitly allowed nor denied by the policy). Change analysis is not supported directly. Zanin and Mancini [ZM04] define a formal model of SELinux policies based on sets, where queries are in terms of set membership, dominance, or non-membership; we hypothesize that this model could be expressed in Datalog, similar to Sarna-Starosta and Stoller.

Work by Kolovski et al. [KHP07] uses Description Logic (DL) to represent XACML policies, which are XML-based access control policies. Description logic is a decidable fragment of first order logic, and is the basis of the web ontology language (OWL). This correspondence makes it easier to reference web elements in policies. Policy analysis services correspond to DL reasoning services (policy inclusion is reduced to concept subsumption, and change analysis is reduced to concept satisfaction). Kolovski et al. use off-the-shelf DL reasoning tools in their implementation, e.g., Pellet (`pellet.owldl.com`).

3.2 Querying Policies Using Model Checking

Guttman et al. [GHRS05] have aims similar to Sarna-Starosta and Stoller in verifying SELinux policies, but query policies using a specialized language (similar to regular expressions) and encoded as a binary decision diagram (BDD) suitable for model checking (by the model checker NuSMV). While superficially different than using logic programming, both approaches essentially encode the policy as a labeled graph using the same underlying encoding and permit querying properties of the graph.

May and Gunter [MGL06] using model checking to analyze policies mandated by the Health Insurance Portability and Accountability Act of 1996 (HIPAA) [hip96]. They formalize these policies as rules in the Harrison-Ruzzo-Ullman (HRU) [HRU76] access control language, and encode the HRU rules in models checkable via the SPIN model checker (`www.spinroot.com`). The strategy is to translate each access control command into a Promela process (the language of SPIN models) which communicates with other processes by named channels. Queries are expressed by constructing sample interactions between principals in the Promela program, and the model checker is used to see whether the interaction succeeds or fails (indicating success or failure, depending on the purpose of the query).

Margrave [FKMT05] is a system for understanding the implications of XACML policies. Kolovski et al. argue that it is less expressive than the description logic-based approach, but it gets much better performance. Like Guttman et al, Margrave uses BDDs to represent policies, which makes for easy comparisons and membership queries; the tool implements its own model checker. Bryans [Bry05] explores reasoning about XACML policies by encoding their elements as *communicating sequential processes* (CSP) and then performing model checking to implement queries. This is similar to the May and Gunter encoding of HIPAA policies. Zhang et al. [ZRG05] encode XACML policies and verification queries in their own modeling and query language, *RW*, and implement queries via model checking.

3.3 Analyzing Cross-Domain Policies

We believe that cross-domain policies can be analyzed techniques similar to those described here. In particular, the key idea is to be able to represent policies as trees or graphs, and then to traverse the graph in ways (using relational or logical queries) that reflect security-related implications. Policies based on trusted declassifiers or extensions thereto have a natural graph-based representation, so we would expect to use such policies as a starting point, and then expand them to include features from role-based policy languages such as RT_0.

4 Conclusions

We have discussed approaches for specifying formal security policies that support downgrading, which is a key operation used in secure cross-domain communications. We have also discussed ways in which formal policies can be modeled and analyzed using logic programming languages and model checkers. We believe that past work in both areas can be extended and improved to support realistic cross-domain security policy construction and validation.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U. S. Government.

References

[BL75] David E. Bell and Leonard J. LaPadula. Computer security model: Unified exposition and Multics interpretation. Technical Report 75-306, ESD, June 1975.

[Bry05] Jery Bryans. Reasoning about xacml policies using csp. In *SWS '05: Proceedings of the 2005 workshop on Secure web services*, pages 28–35, 2005.

[CM04] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 198–209, New York, NY, USA, 2004. ACM Press.

[FKMT05] Kathy Fisler, Shriram Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *Proc. ACM SIGSOFT Conference on Software Engineering*, pages 196–205, 2005.

[GHRS05] Joshua D. Guttman, Amy L. Herzog, John D. Ramsdell, and Clement W. Skorupka. Verifying information flow goals in Security-Enhanced Linux. *Journal of Computer Security*, 13(1):115–134, 2005.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, April 1982.

[GM84] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–86. IEEE Computer Society Press, April 1984.

[hip96] Health insurance portability and accountability act of 1996. Technical Report Public Law 104-191, H.R. 3103, Health Resources and Services Administration, 1996.

[HKMH06] Boniface Hicks, Dave King, Patrick McDaniel, and Michael Hicks. Trusted declassification:: high-level policy for a security-typed language. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 65–74, New York, NY, USA, 2006. ACM Press.

[HRC+07] Boniface Hicks, Sandra Rueda, Luke St. Clair, Trent Jaeger, and Patrick McDaniel. A logical specification and analysis for SELinux MLS policy. In *Proc. ACM symposium on Access control models and technologies*, 2007.

[HRU76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[KHP07] Vladimir Kolovski, James Hendler, and Bijan Parsia. Analyzing web access control policies. In *Proc. W3C Annual World Wide Web Conference*, May 2007. To appear.

[LM03] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust-management languages. In *Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages (PADL 2003)*, pages 58–73, January 2003.

[LMW02] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society Press, May 2002.

[LMW05] Ninghui Li, John C. Mitchell, and William H. Winsborough. Beyond proof-of-compliance: Security analysis in trust management. *Journal of the ACM*, 52(3):474–514, May 2005.

[LZ05a] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. 32nd ACM Symp. on Principles of Programming Languages (POPL)*, 2005.

[LZ05b] Peng Li and Steve Zdancewic. Unifying confidentiality and integrity in downgrading policies. In *Proc. of Foundations of Computer Security Workshop*, 2005.

[Mal07] Pasquale Malacaria. Assessing security threats of looping constructs. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 225–235, 2007.

[MGL06] Michael J. May, Carl A. Gunter, and Insup Lee. Privacy APIs: Access control techniques to analyze and verify legal privacy policies. In *19th IEEE Computer Security Foundations Workshop (CSFW)*, July 2006.

[MNZZ01] Andrew C. Myers, Nathaniel Nystrom, Lantian Zheng, and Steve Zdancewic. Jif: Java + information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[MSZ04] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing Robust Declassification. In *Proc. of 17th IEEE Computer Security Foundations Workshop*, pages 172–186, Asilomar, CA, June 2004. IEEE Computer Society Press.

[SHTZ06] Nikhil Swamy, Michael Hicks, Stephen Tse, and Steve Zdancewic. Managing policy updates in security-typed languages. In *Proc. of 19th IEEE Computer Security Foundations Workshop*, pages 202–216, July 2006.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[SM04] Andrei Sabelfeld and Andrew C. Myers. A model of delimited information release. *Lecture Notes in Computer Science*, 3222:174–191, 2004.

[SS05] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proc. of the 18th IEEE Computer Security Foundations Workshop*, pages 255–269, Los Alamitos, CA, USA, 2005.

[SSS04] Beata Sarna-Starosta and Scott D. Stoller. Policy analysis for Security-Enhanced Linux. In *Proc. of 2004 Workshop on Issues in the Theory of Security*, pages 1–12, April 2004.

[ZM01] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proc. of 14th IEEE Computer Security Foundations Workshop*, pages 15–23, Cape Breton, Canada, June 2001.

[ZM04] Giorgio Zanin and Luigi Vincenzo Mancini. Towards a formal model for security policies specification and validation in the SELinux system. In *Proc. 9th ACM symposium on Access control models and technologies*, pages 136–145, 2004.

[ZRG05] Nan Zhang, Mark D. Ryan, and Dimitar Guelev. Evaluating access control policies through model checking. *Lecture Notes in Computer Science*, 3650:446–460, 2005.